



LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Havery Mudd 2014-2015 Computer Science Conduit Clinic Final Report

G. Aspesi, J. Bai, R. Deese, L. Shin

May 21, 2015

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.



Computer Science Clinic

Final Report for
Lawrence Livermore National Laboratory

Conduit: Scientific Data Exchange for HPC Simulations

May 7, 2015

Team Members

George Aspesi
Justin Bai
Linnea Shin
Rupert Deese (Project Manager)

Advisor

Robert Keller

Liaisons

Cyrus Harrison
Adam Kunen
Brian Ryujin

Abstract

Conduit, a new open-source library developed at Lawrence Livermore National Laboratories, provides a C++ application programming interface (API) to describe and access scientific data. Conduit's primary use is for in-memory data exchange in high performance computing (HPC) applications. Our team tested and improved Conduit to make it more appealing to potential adopters in the HPC community. We extended Conduit's capabilities by prototyping four libraries: one for parallel communication using MPI, one for I/O functionality, one for aggregating performance data, and one for data visualization.

Contents

| | |
|---|------------|
| Abstract | iii |
| 1 Introduction | 1 |
| 1.1 Sponsor Background | 1 |
| 1.2 Project Background | 1 |
| 1.3 Problem Objectives and Components | 2 |
| 2 Conduit | 3 |
| 2.1 Conduit Nodes | 3 |
| 2.2 Constructing Nodes | 4 |
| 3 Conduit MPI Library | 5 |
| 3.1 Motivation | 5 |
| 3.2 Architecture | 5 |
| 3.3 Supported Functions | 6 |
| 3.3.1 Send and Recv | 6 |
| 3.3.2 ISend and IRecv | 6 |
| 3.3.3 Reduce | 7 |
| 3.3.4 Allreduce | 7 |
| 3.3.5 Waitsend and Waitrecv | 7 |
| 3.3.6 Waitallsend and Waitallrecv | 7 |
| 3.4 Testing | 8 |
| 3.5 Future Work | 8 |
| 4 Conduit I/O Library | 9 |
| 4.1 Motivation | 9 |
| 4.2 Architecture | 9 |
| 4.3 Storage Paradigms | 9 |
| 4.3.1 Cold Storage | 9 |
| 4.3.2 Object-mapping | 10 |

| | | |
|----------|--|-----------|
| 4.4 | Testing | 10 |
| 4.4.1 | Cold Storage | 11 |
| 4.4.2 | Object-mapping | 11 |
| 4.5 | Future Work | 11 |
| 4.5.1 | Further testing | 11 |
| 4.5.2 | Parallelism | 11 |
| 4.5.3 | Extending object-mapping | 12 |
| 4.5.4 | Reading object-mapped files | 12 |
| 4.5.5 | Generalized structure-aware storage | 12 |
| 4.5.6 | Other data formats | 12 |
| 5 | Timer Trees | 13 |
| 5.1 | Structure | 13 |
| 5.2 | Architecture | 14 |
| 5.3 | Parallel Reduction | 15 |
| 5.4 | Benchmarking | 16 |
| 5.5 | Future Work | 16 |
| 5.5.1 | Multithreading | 16 |
| 5.5.2 | Recursion | 16 |
| 5.5.3 | Triple-underscores | 17 |
| 5.5.4 | Depth | 17 |
| 6 | Visualizer | 19 |
| 6.1 | Motivation | 19 |
| 6.2 | Features | 19 |
| 6.2.1 | Tree view | 20 |
| 6.2.2 | Treemap view | 21 |
| 6.2.3 | Inspector | 22 |
| 6.2.4 | Settings | 22 |
| 6.3 | Architecture | 22 |
| 6.3.1 | Model | 23 |
| 6.3.2 | View | 23 |
| 6.3.3 | Controller | 23 |
| 6.4 | Testing | 24 |
| 6.5 | Future Work | 25 |
| 6.5.1 | Required Testing | 25 |
| 6.5.2 | Special Visualization of Timer Trees | 25 |
| 6.5.3 | Integration with Conduit I/O | 25 |

| | | |
|----------|---|-----------|
| 7 | Conclusions | 27 |
| 7.1 | Deliverables | 27 |
| 7.1.1 | MPI Library | 27 |
| 7.1.2 | I/O Library | 27 |
| 7.1.3 | Timer Trees | 27 |
| 7.1.4 | Visualizer | 28 |
| 7.2 | Conclusion | 28 |
| A | Glossary | 29 |
| B | Technology Background | 31 |
| B.1 | MPI | 32 |
| B.2 | LULESH | 33 |
| B.3 | Silo | 34 |
| B.4 | VisIt | 34 |
| C | Documentation | 35 |
| C.1 | Conduit MPI Library | 36 |
| C.1.1 | Functions | 36 |
| C.1.2 | Structures | 40 |
| C.2 | Conduit I/O Library | 41 |
| C.2.1 | Functions | 41 |
| C.2.2 | Additional functions for future PMPIO functionality | 44 |
| C.2.3 | Schemas for silo_structured_save | 46 |
| C.3 | Timer Trees | 48 |
| C.3.1 | Usage | 48 |
| C.3.2 | Functions and Variables | 48 |
| C.4 | Visualizer | 50 |
| C.4.1 | Usage | 50 |
| C.4.2 | Components | 50 |
| | Bibliography | 55 |

Chapter 1

Introduction

1.1 Sponsor Background

Lawrence Livermore National Laboratory (LLNL) is a research facility for national security primarily funded by the United States Department of Energy. LLNL is well known for its work in nuclear security, ongoing nuclear fusion experiments at its National Ignition Facility, and for its advances in high performance computing (HPC). LLNL's collection of HPC hardware includes IBM Sequoia which, with more than 16 petaFLOPS of computing power, is ranked the fastest computer in the world for data-intensive applications [13]. Scientists use LLNL's hardware to run massively parallel physics simulations, and are always looking for ways to simplify the process of writing parallel programs.

1.2 Project Background

Scientists at LLNL have developed a new, open source library called Conduit. Conduit provides a flexible and sensible means of describing and storing scientific data in memory in C++ applications. Conduit provides a lightweight format similar to Javascript Object Notation (JSON) for managing scalars and arrays of explicit precision in hierarchical structures and is self-introspective. Conduit is envisioned as an easy and intuitive library for data description and exchange in HPC programs.

More information on Conduit can be found in Chapter 2. Peripheral technologies used in this project are described in the glossary and the appendixes.

1.3 Problem Objectives and Components

The LLNL 2014-2015 Clinic team was asked to expand Conduit's potential for HPC data handling. By providing frequently used operations through Conduit's API, and demonstrating its use in demo applications provided by LLNL, this project will help encourage adoption of Conduit amongst scientists in the HPC community.

To accomplish this, we added four major features to Conduit:

- Chapter 3: A library of MPI wrappers for handling parallel data
- Chapter 4: A library of IO methods for common file formats and workflows
- Chapter 5: A performance benchmarking library in the form of timer trees
- Chapter 6: A visualizer for Conduit objects

Chapter 2

Conduit

At LLNL, scientists work with massively parallel code on some of the largest supercomputers in the world. Current programming paradigms do not always scale well to the level of parallelism being used at LLNL. Conduit, a C++ data-handling library, was developed by scientists at LLNL in order to address the limitations of existing methods of data handling and exchange at this scale. Conduit is designed to make data in memory easier to describe, access, and manipulate.

2.1 Conduit Nodes

The Conduit library defines the `Node` class, which is used to organize data in a tree structure. Instances of the `Node` class (called "Nodes") can be widely classified into two groups: leaf Nodes that hold data, and non-leaf Nodes that point to other Nodes.

Leaf Nodes support a range of data types: all commonly used widths of integer and float, booleans, and `std::strings`. Arrays and `std::vectors` of numerical types are also supported. Depending on how they are assigned data, leaf Nodes can either own the data they hold, or refer to existing data in memory. Through a leaf Node, one can access the data's type, endianness, length, and offset in memory, in addition to the data itself. All of this supplementary information is stored in the Node's schema, a JSON (Javascript Object Notation) formatted string.

Non-leaf Nodes cannot hold data, but can contain any number of children Nodes, which are differentiated by string labels. The schema of a non-leaf node is a dictionary structure which contains the schemas of all child Nodes, under their corresponding labels. The children of a non-leaf

Node (and their children, if applicable) can be accessed via their labels using the bracket (`[]`) operator, which the Node class overloads to accept strings. This functionality of non-leaf nodes is similar to Python dictionaries or Javascript objects.

2.2 Constructing Nodes

Nodes can be constructed in two main ways: from scratch using a dynamic setter functionality, or from a previously existing Node, using the Node's schema and its compressed data. To facilitate the second use case, Nodes can copy their data into contiguous memory and return it as a bytestream, along with their complete schema.

Chapter 3

Conduit MPI Library

3.1 Motivation

Parallel programming is often required in high-performance computing, in order to minimize the running time. This becomes necessary in physics simulation programs, such as those developed at LLNL. One such way of achieving this is a paradigm called MPI (Message Passing Interface). It allows for the functionality of sending messages between processes. MPI allows a user to instantiate several different processes, which share no data, to work on a single problem. Each process belongs to one or more communicators, which serve as message sending channels, and has a unique identification number known as a rank within each communicator. MPI functions allow processes to communicate information to one another, allowing for coordination and compilation of information. More information on MPI can be found in Appendix B.1.

One of the additions to the Conduit ecosystem was an MPI wrapper library, hereafter referred to as the Conduit MPI Library, or Conduit MPI. It allows for both simplified use of common MPI functions, as well as allowing a greater degree of flexibility by utilizing Conduit Nodes.

3.2 Architecture

The functions in Conduit MPI serve as wrapper functions, leveraging the existing functions but allowing the data being sent to be a Conduit Node, which allows for not only a simplification of usage for the functions but also allows for a great amount of versatility. These functions can be used in place of their non-Conduit counterparts, allowing LLNL to place them easily

into their workflow. Currently, Conduit MPI provides the MPI functions `Send`, `Recv`, `ISend`, `IRecv`, `Reduce`, and `Allreduce`, and additional support functions.

3.3 Supported Functions

3.3.1 Send and Recv

`Send` and `Recv` are a pair of functions that allow two processes to transfer data from one to the other. The process that wishes to send data uses the function `Send`, taking as input the Conduit Node, the rank of the recipient process, the tag, and the communicator over which to send. The receiver uses `Recv`, taking as input the Conduit Node into which they want data placed, the sender, the tag, and the communicator. These are blocking send and receive, meaning that the processes will not continue until both nodes reach the send and receive, at which point the transfer occurs. This function works by making a number of standard MPI sends, firstly an array of two integers representing the length of the Node's schema and data, then the schema as an array of characters, and finally the data, also as an array of characters. When this data is received, it is constructed into a Node using a generator with the schema and walking through the data.

3.3.2 ISend and IRecv

`ISend` and `IRecv` operate similarly to `Send` and `Recv` above. The primary difference is that `ISend` and `IRecv` are not blocking. Each takes an additional input, a Conduit MPI Request which contains pointers to heap-allocated Conduit Nodes where data is stored prior to transfer. These requests are non-blocking, which means the transfer does not occur immediately and both processes can continue operation until the results are required. At this point, the processes call two functions, `Waitsend` and `Waitrecv`, which complete the transfer. Furthermore, an additional constraint exists in this version in that the receiving process is required to have a Conduit Node with a matching schema of the sent Conduit Node, as this send receive pattern only sends the data, and the receiver updates the Conduit Node with the new data.

3.3.3 Reduce

Reduce operates by reducing a series of Conduit Nodes from several different processes into a single Conduit Node. The resulting Node will have the same structure as each of the constituent Nodes, and each element of the reduced Node will be the reduction of each element in the constituent Nodes, reduced according to a binary MPI operation taken as input. This function has several constraints. Firstly, the nodes must each have the same structure and schema. Otherwise, not every element will have a likewise element to combine with. Secondly, all elements in the Conduit Node must be of the same type, for simplification. The function still allows for any structure in the Nodes, however. The inputs to these functions take as input the Node to be reduced, the Node where the result is to be placed, the index of the root process, where the final reduced Node will accumulate, the MPI datatype for the Nodes, the MPI operation they wish to be performed, and the communicator.

3.3.4 Allreduce

Allreduce functions exactly as Reduce, except that instead of the result being placed in the root process, the result is distributed to all participating processes. The inputs are the same, except it doesn't take a root index.

3.3.5 Waitsend and Waitrecv

Waitsend and Waitrecv are used in conjunction with ISend and IRecv. Because ISend and IRecv are non-blocking, it is possible for Waitsend and Waitrecv to be run and the actual send not complete until a later time. These functions, when called, block execution until the other process has also called wait. Using ConduitMPIRequest objects as receipts of previous ISends and IRecvs, Waitsend and Waitrecv then complete the send, copying any relevant data over, and freeing any objects that remain.

3.3.6 Waitallsend and Waitallrecv

Waitallsend and Waitallrecv act like Waitsend and Waitrecv, except they take arrays of ConduitMPIRequest objects, and wait until all sends represented by these requests complete before continuing execution.

3.4 Testing

The testing of this library included a GTest (Google Test) testing class with separate unit testing for each function [1]. Edge cases that were tested included Nodes that had their data set externally, meaning the Node leaf points to external data instead of containing the data itself. Other edge cases included Nodes with complex data types and structures, and reductions with many different reduction operations, in the case of Reduce and Allreduce. Furthermore, Conduit MPI was inserted into LULESH, a "mini-app" designed for testing features on high performance computing physics simulations. More information on LULESH can be found in Appendix B.2. Specifically, it was inserted to prove its ability to send data from several locations in a single message to different processes.

3.5 Future Work

Future work for the library would involve further increase in functionality. The functions included in the library are those used most often in high-performance computing at Lawrence Livermore National Laboratory and thus of most use to our client, but other functions, such as Gather, Scatter, Broadcast, and Alltoall, can easily be added to the library in the future. Additional testing and integration into LULESH would accompany these as testing, and as a proof that the library doesn't strongly negatively affect running time for physics simulation programs.

Chapter 4

Conduit I/O Library

4.1 Motivation

Easy storing and sharing of data is an important usability feature that scientists will expect before actively adopting Conduit. In order to facilitate I/O functionality in Conduit, we added a new library that interfaces with Silo to implement two common workflows at LLNL.

4.2 Architecture

Silo is a library used to read and write scientific data to binary files. It emulates a file directory structure and has built in support for many of the specialized data structures used in physics simulations. For more information on Silo, see B.3.

The Conduit I/O Library is a modular library with functions for dynamically determining what underlying library to use for writing to disk. Currently the library only supports using the Silo library. The work here outlines the functionality for two storage paradigms, a compressed format, and a structured format. These formats were designed in conjunction with the liaisons in order to satisfy internal use cases and requirements.

4.3 Storage Paradigms

4.3.1 Cold Storage

Cold storage is a simple, compressed storage format. The new Conduit I/O library exposes several simple functions for working with Silo files in cold

storage format. To write to disk, the application passes in a reference to the Node to be stored, and optionally an existing Silo file to update or the name of such a file. The resulting Silo file contains two data objects:

- A string representing the schema of the stored Conduit file
- The byte data of the compressed Conduit node

To read from disk, the application passes in a Silo file, and optionally a Conduit Node to populate with the data. Conduit parses the stored schema and uses it to understand the byte data, and constructs the Node.

This format has the additional benefit of accepting arbitrary Nodes, regardless of structure or data members; as long as it is a valid Conduit Node, it is eligible for this storage format.

Cold storage was designed to use existing Conduit functionality for serializing data in as lightweight a structure as possible.

4.3.2 Object-mapping

Object-mapped storage files leverage the Silo database's support for structured, hierarchical data. These files can also use Silo's native data structures that are specialized for scientific data, such as physics meshes (see B.3).

The resulting files can be inspected using programs such as Silex (see B.3), our included visualizer (section 6), or opened using LLNL's interactive physics visualizer, VisIt (see B.4).

To create an object-mapped file, the application must pass in a Conduit Node that conforms to a specific Node structure defined in the documentation in section C.2.3. The library can translate data matching this schema to a subset of Silo mesh structures.

Object-mapped files were initially developed based off of existing Silo functionality in a provided demo application by LLNL. First we designed the library to create Silo files that were identical to the original ones. Later, we iterated on this functionality to generalize to all physics applications working with a certain subset of datatype and structures, and to better utilize Conduit's functionality for parsing and writing the data.

4.4 Testing

Unit tests were developed while creating the Conduit I/O library. Additionally, results were manually tested.

4.4.1 Cold Storage

Cold storage files written by the library were tested by manually inspecting the schema and byte data of resulting files to ensure they contained the expected data. Unit tests ensure that data is written with no errors, and ensuring that new Nodes are correctly populated from file using the read functions.

4.4.2 Object-mapping

Originally, object-mapped files were tested via manual inspection and provided LLNL command line utilities designed for comparing Silo files, in order to ensure that the created files match the old functionality of a provided LLNL application. As the functionality evolved to be more generalized, files were tested via manual inspection and in VisIt (B.4) to ensure that the correct data members exist in the correct formats.

4.5 Future Work

Future work on the Conduit I/O library has a number of potential paths.

4.5.1 Further testing

The Silo functionality and library does not contain a very comprehensive set of unit tests, especially for the object-mapping paradigm. Tests should be written to help ensure that future iterations do not break the expected functionality.

4.5.2 Parallelism

Silo is not natively a parallel application. However, Silo includes a library for writing parallel data using a paradigm called Poor Man's Parallel IO (PMPIO). PMPIO allows serial I/O to handle parallel data by labelling compute nodes and managing their write operations via handoff of a simple data structure called a "baton" [11].

Parallel writes via PMPIO can be easily added to the current implementation by having the relevant functions be aware of which compute node is calling the write. The code for this feature has been written, in a new set of files named `conduit_silo_mpi`. However, this functionality has not been integrated into the compile target or tested.

4.5.3 Extending object-mapping

The object-mapped storage files can be expanded to translate to a greater number of Silo structures, such as data types and mesh geometries that are not currently supported. Adding cases to the library function should require little effort as it is mostly just extra cases with different parameters for the write calls to the Silo file.

4.5.4 Reading object-mapped files

Object-mapped storage files can be read into Nodes without much additional coding effort. However, this functionality may lose data, especially the structure of the original Conduit Node, and type details such as endianness and width, as this information is not retained during the translation required to map Conduit Node objects to native Silo data types and structures. Also, the usefulness of such functionality has not been established.

4.5.5 Generalized structure-aware storage

Currently, structured data can only be stored when passed to the library following a defined schema. More generalized functionality could be used to map an arbitrarily constructed Node to a Silo file. This functionality would be an intermediate between the flexibility of cold storage, and friendly, human readable object-mapped files. However, while such a file format would be able to map primitive data types between Conduit and Silo (e.g. determining integer versus floating point and data widths), it would not be capable of mapping to complex Silo data structures such as meshes (see B.3) as it cannot determine how the mesh is structured in the Conduit Node. Also, the usefulness of such functionality has not been established.

4.5.6 Other data formats

Although Silo files are a powerful and appropriate format for storing Nodes on disk, it is feasible to add further functionality for other file formats or data encodings.

Chapter 5

Timer Trees

We implemented a timing library using Conduit that allows developers to conveniently collect performance data about their programs. A key use case is to quickly identify poorly performing functions and processes to focus attention on them. The user can put a one-line macro at the beginning of functions to be timed, and the library will collect data in the back-end. The library produces a tree of Conduit Nodes that summarizes the program's execution of all timed functions. When using MPI, each process generates its own timing tree. To quickly analyze these, we have provided a reduction function that combines these trees into one, keeping track of metrics such as the minimum (time spent), maximum, average, which computers gave those values, and how many processes executed a particular path.

5.1 Structure

A program's timing tree is stored in a static Node that refers to the root of the tree. The root represents the main function, and child Nodes represent function calls, accounting for nesting. Each node holds performance data for the corresponding portion of the program. For example, if `foo` and `bar` are timed functions and `main` calls them sequentially, then the tree will look like Figure 5.1. However, if `foo` also calls `bar`, then the tree will look like Figure 5.2.

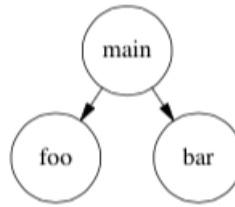


Figure 5.1 An abstract representation of a timer tree. Here `main` calls both `foo` and `bar`.

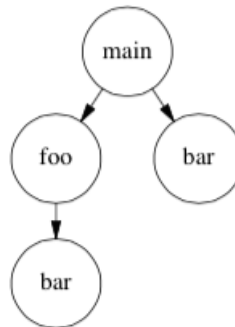


Figure 5.2 An abstract representation of a timer tree. Here `main` calls both `foo` and `bar`, and `foo` also calls `bar`.

Note that while the two `bar` nodes in Figure 5.2 refer to the same function, they keep track of separate paths in the program's execution. In addition, untimed functions are not shown in the timer tree, so the parent-child relationship implies an *indirect* call, rather than a direct call. For example, it's possible that `foo` calls `baz` which calls `bar`, but if `baz` is untimed, then it doesn't appear in the timer tree.

5.2 Architecture

Timing is handled by the block timer defined in the `BlockTimer` class. It uses static variables to keep track of a process' execution (a static variable is shared by all instances of the class). Timing is started and stopped by the construction and destruction of a `BlockTimer` instance. When a block timer is constructed, a timer is started; when that block timer is destructed, that timer is stopped and the time elapsed is recorded. This allows the user to time any function by instantiating a block timer at the beginning of the

function body. The result is a static Node that contains the entire timing tree. This can be printed or dumped to a file for transfer or visualization.

Note that this architecture cannot directly time the `main` function, since the timing Node needs to either be printed or dumped before the `main`'s block timer would be destructed. To include this data, one must manually handle `main`'s timing and put the data into the timing node.

5.3 Parallel Reduction

When using MPI, each process will generate its own timer tree. We have provided a reduction function to condense these into a single tree. For each Node the reduction keeps track of the minimum (time spent), maximum, average, which computers gave those values, and how many processes executed a particular path. However, other basic reduction operations can be added with only a little effort (see documentation). Inter-process reduction uses blocking sends and receives from the Conduit MPI library to communicate.

Figure 5.3 shows part of how a maximum reduction of two timer trees works. Between two corresponding Nodes, the maximum time is chosen and the original ID (MPI rank of the process) is recorded. When trees have differing paths, the union of the paths is incorporated into the result.

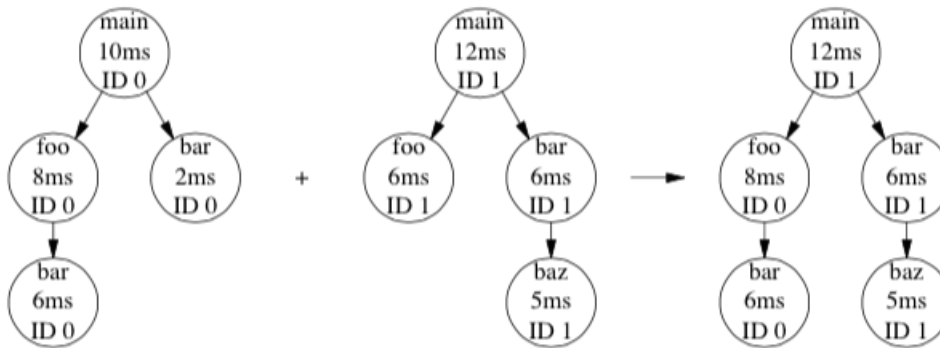


Figure 5.3 An abstract representation of a maximum reduction of two timer trees. The original trees are shown at the left and middle, and the reduced tree is shown at the right.

5.4 Benchmarking

The timing class was used in LULESH for basic benchmarking. All versions of LULESH for testing were compiled with MPI but without OpenMP. Three versions were benchmarked: an out-of-the-box version of LULESH, a timed version of LULESH, and a timed version of LULESH with periodic Conduit dumps of the LULESH state. Each was run 5 times using `srun` on LLNL's machine Cab for each of 1, 8, and 27 processes [10]. All versions used the same block timers and timed all of the functions down to 4 levels of nesting (where `main` is the 0th level). Timing these functions added less than 5% of the untimed version's runtime, and executing Conduit dumps to files took roughly 1 additional second per process.

5.5 Future Work

Some aspects that could be modified in the future include the following.

5.5.1 Multithreading

The timing class is not compatible with multithreading because its static data members need to know where the program is in its execution. This is a grave issue when threads can be in different functions simultaneously.

Since multithreading is an important aspect of high-performance computing, it would be useful to extend the library to support it. This may not be easy to do elegantly: either the core architecture of the timing class would have to be changed, or the library would need to handle multithreaded timing as a separate case.

5.5.2 Recursion

Recursion presents a visibility issue because the same function will be called over and over again, possibly leading to very deep timing trees that don't easily convey particularly useful information, and likely making the program much more inefficient (if a lot of recursive calls are made). Adding an option to treat a timer as recursive (to not add depth for a recursive call, essentially) may make visualization much better. Recursive aspects could also have other information, like min and max times (perhaps more useful for tail-recursive functions). It is unclear, however, how tail recursion would interact with the `BlockTimer` destructor; however, it is likely they will still have the problem.

5.5.3 Triple-underscores

The built-in node names, such as `___min` and `___children`, have triple underscores in front of them. This is to prevent name conflicts with actual functions in programs, but may impede user visibility. A solution that eliminates the need for triple underscores would likely improve readability a little.

5.5.4 Depth

The library has an option for depth control. While this may be useful for trimming output, it adds conditional checks to the constructor and destructor, potentially limiting efficiency a little bit. This hasn't been deemed critical because basic timing is within the noise anyways and CPU branch prediction may limit some inefficiency. However, the depth control may be unnecessary if users only time a few functions and do not need the execution structure.

Chapter 6

Visualizer

6.1 Motivation

Many of the software libraries in common use at LLNL are supported by an ecosystem of tools and applications that abstract the use of the library away from the command line. These tools enrich interaction with the library and facilitate common workflows, making the libraries themselves more appealing.

The Conduit Visualizer supports the use of Conduit, increasing its usefulness and attractiveness to potential adopters. The Visualizer is designed to allow users to explore data stored in Conduit Nodes. The Visualizer can open any Conduit Node that has been written out to disk. Specifically, it supports two use cases. The first is exploration: using the Visualizer to gain familiarity with, or insights into, data. The second is inspection: using the Visualizer to conveniently access and check data, when the user is already familiar with the data.

To support the exploratory use case, the Visualizer contains graphical components to intuitively represent the data in the Node: a tree and a treemap. To support the inspection use case, the Visualizer has powerful search functionality, and a value table that can be used to read values out of any data member in the Node.

6.2 Features

The following sections describe the key features and functionality of the Visualizer. The complete Visualizer is shown in Figure 6.1. For more detailed

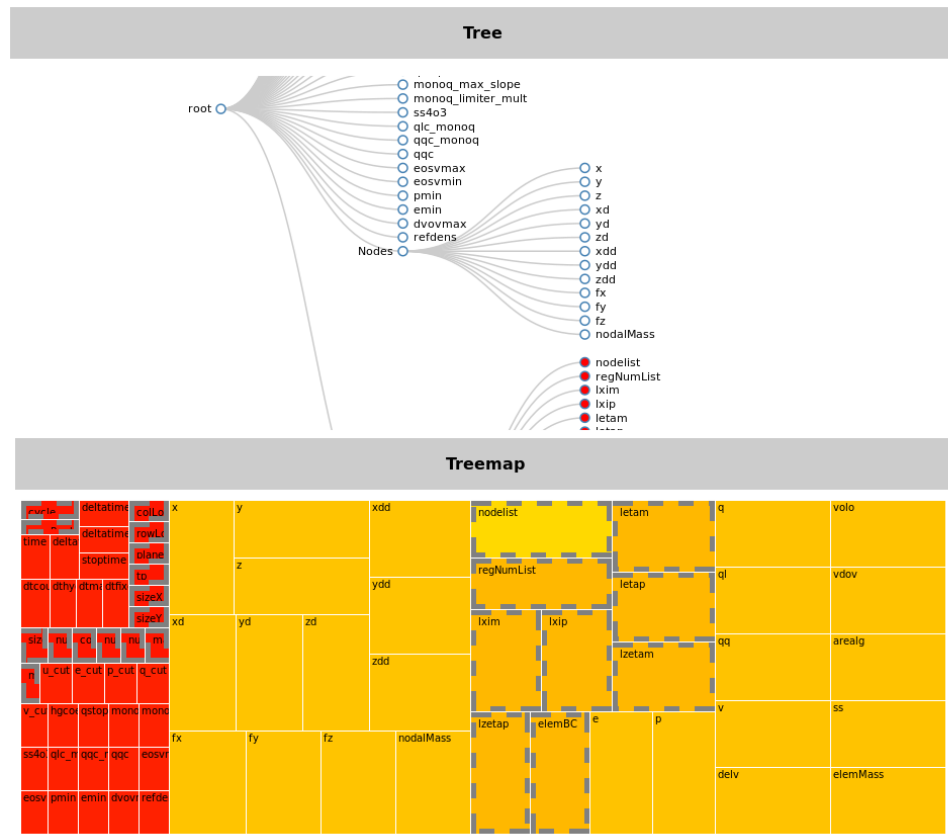


Figure 6.1 The Visualizer.

images and descriptions of each component, see the Visualizer documentation in Appendix C.4.

6.2.1 Tree view

The tree view presents a hierarchical representation of the Node's structure. The tree view is displayed using the Reingold-Tilford layout to maximize readability and compactness [14]. Each data member is labeled in the tree with its name in the Node's schema. The tree view can be zoomed and panned, to accommodate arbitrarily large Nodes. Data members in the tree can be selected with the mouse, either to collapse or expand their children, or to bring up further information about the selected data member. Children in the tree can be ordered alphabetically, by size, or as they appear in the

Inspector

Node search

int

○name

●type

Filter

Search results

| Path | Type | Length | Size |
|-------------------|-------|--------|----------|
| / | | | |
| /numNode | int32 | 1 | 4 B |
| /maxPlaneSize | int32 | 1 | 4 B |
| /maxEdgeSize | int32 | 1 | 4 B |
| /Elems/nodelist | int32 | 216000 | 843.8 kB |
| /Elems/regNumList | int32 | 27000 | 105.5 kB |
| /Elems/ixim | int32 | 27000 | 105.5 kB |

Value inspector

| Index | Value |
|-------|-------|
| 98740 | 14117 |
| 98741 | 14118 |
| 98742 | 14149 |
| 98743 | 14148 |
| 98744 | 13157 |
| 98745 | 13158 |

Settings

Treemap scale

○bytes

●log₁₀₂₄(bytes)

○equal

Node sorting

○offset

○alphabetical

○size

Quit Visualizer

schema, according to the user’s preference.

6.2.2 Treemap view

A treemap is an alternate format for displaying hierarchical data, one that incorporates the size of data. In a treemap, each data member is represented as a rectangle, nested inside its parent. The area of each rectangle corresponds to some measure of the size of the data member. In the Visualizer’s treemap, the rectangles are sized relative to their size in memory or on disk. This size can be shown linearly, calculated on a logarithmic scale, or ignored (all data members have equal area). Each rectangle is labeled with the schema name of its corresponding data member. Finally, each rectangle is colored according to its size on a logarithmic scale.

The rectangles visible in the treemap at any given time reflect how the

tree view is expanded: data members that are visible in the tree view, and have no children (they are either leaves or their children are collapsed) get rectangles in the treemap. This allows the user to adjust the treemap to show the relations she is specifically interested in. Rectangles in the treemap can be selected with the mouse to bring up further information on their corresponding data member.

6.2.3 Inspector

The inspector provides a search interface, a table to view search results, and a table to show the values of any selected data member. The search interface can query by size and datatype, and the user can specify partial or exact matches. The table that shows the results displays the name, datatype, length, and size in memory of each match. In addition, all matches are highlighted in both the tree and treemap views.

The value table shows the actual values stored in the selected data member. Data members can be selected through interaction with the tree or treemap view, or an exact search match. The value table displays values of any datatype, and supports arbitrarily large arrays.

6.2.4 Settings

A settings panel provides toggles for treemap scaling modes and tree and treemap sorting. The user can choose the sorting order of nodes in the Tree view: default, alphabetically, or by size. The user can choose the way size is displayed in the treemap: on a linear scale, a log scale, or equal sizes for all data members. The settings panel also includes the "Quit Visualizer" button, which terminates the visualizer server.

6.3 Architecture

The Conduit Visualizer is a client-server application. The server is a C++ library that is used to launch the Visualizer. It leverages two C++ libraries:

- Conduit
- mongoose [7]

The client is a web application built using a number of languages and libraries:

- HTML/CSS [6]
- Javascript [3]
- D3.js [4]
- Fattable.js [12]
- Pure.css [15]

A convenient way to understand the structure of the Visualizer is through a loose Model-View-Controller (MVC) paradigm. The model is a Conduit Node and the data it contains; the view is the web application presented to the user. The controller sits between the two, generating and modifying the view using the model, in response to input from the user via the view.

6.3.1 Model

The model is a C++ pointer to a Node provided by the user as an argument when the Visualizer is launched. The Visualizer does not own the model, which is a Node existing somewhere else in the calling program's code. The Visualizer simply accesses the model through the server-side C++ component of the controller, which interfaces with the rest of the Visualizer.

6.3.2 View

The view is constructed in HTML and CSS. Pure.css is used to abstract the creation of the overall grid layout, elegant user inputs, and the search table. D3.js is used to generate the complicated HTML and vector graphics used for the tree, treemap, and search table. D3.js is also used for event handling, which binds the view to the controller. Finally, the Fattable.js library is used to create the value table, which requires special optimization to handle tens of thousands of entries.

6.3.3 Controller

The controller consists of:

- a C++ library that serves the client web page, responds to HTTP requests from the client, and queries the model.
- Javascript code defining the client-side behavior of the program.

The C++ library exposes a single function that takes a Node as its only argument. This function halts program execution, and uses the mongoose library to serve the client web application to `localhost:8080`. It defines an event handler that augments the mongoose web server to respond to HTTP POST requests from the client by querying the model. Two endpoints are defined: one returns the model's JSON schema, and the other returns the data contained at a leaf Node specified in the request.

The controller's client-side functionality is separated into a number of Javascript classes using an object-oriented programming paradigm. The tree and treemap, both tables, and utility functions, are all in separate Javascript classes. An overarching Visualizer class creates and manages instances of all these classes. An instance of this class is in turn created by a short Javascript program which runs when the web page loads.

As previously mentioned, the controller responds to events in the view using D3.js event handling. D3.js somewhat subverts the MVC paradigm by binding the model directly to elements of the view. These direct bindings are what makes it possible to create complex structures like the tree and treemap at a high level of abstraction. The various controller classes employ and modify this "out-of-the-box" functionality of D3.js, by telling D3.js how to manipulate the view on a per-datum basis according to the changes in the view that are necessary. Events in any component of the Visualizer are relayed to the main Visualizer class. This class coordinates any resultant modifications to the view through delegation to instances of the other classes.

In addition to constructing the view for the value table, the Fatable.js library provides a separate controller for the value table, which is linked to data supplied by the model, according to what data needs to be displayed. This controller handles the dynamic creation and destruction of HTML table cell elements, to create the illusion of scrolling while keeping the actual number of rendered cells to a minimum to improve performance.

6.4 Testing

Testing of the Visualizer proceeded through several avenues. Design feedback on features, user interface and interaction, and aesthetics, was obtained from members of the team, our advisors, and most importantly the liaisons. Testing for correctness of the displayed data was conducted by integrating the Visualizer with LULESH.

6.5 Future Work

The current Visualizer should be considered as a proof-of-concept or prototype. It may be suitable for use by developers. To improve upon the Visualizer, thorough testing is necessary.

6.5.1 Required Testing

The Visualizer has not been rigorously tested. Thorough testing is needed to identify potential bugs and performance constraints of the program. We recommend two avenues for testing.

1. Performance on large Conduit Nodes. While the UI and UX elements are designed to support data of any scale, the browser itself has limitations (particularly memory constraints) that have not been explored.
2. Support for unusual Conduit Node Schemas. In theory all Conduit Nodes are supported, but prudence requires that we remain skeptical until a larger range of possible Conduit Schemas are tested.

6.5.2 Special Visualization of Timer Trees

Timer trees contain a specific type of data in a predictable format. To better display this data to users of both timer trees and the Visualizer, the Visualizer could be augmented to detect Nodes in timer tree format and display them using additional visual paradigms. For example, a modified treemap (or several treemaps at once) could be used to show the relative execution times of the various timed functions. This visualization could also include maximum, minimum, and average runtimes if the timer tree is the result of a parallel reduction.

This additional functionality could be added via the client-side Javascript, HTML, and CSS that implement the view and controller. Implementing it would require a working knowledge of HTML and CSS, fluency in Javascript, and a strong understanding of D3.js.

6.5.3 Integration with Conduit I/O

For convenience, Conduit I/O could be integrated into the Visualizer. In particular, two use cases are important.

Firstly, the Visualizer could be used to save a Node to disk, in either of the supported Conduit I/O formats. This would allow users to save

interesting data on the fly. Users would also be able to export and visualize their data in VisIt during runtime, by using Conduit I/O's object-mapped storage.

Secondly, the Visualizer library could be extended to include a stand-alone executable that would read Conduit Nodes from disk and visualize them. This would save users the trouble of having to write and compile their own C++ program to achieve the same result.

Chapter 7

Conclusions

7.1 Deliverables

Our final deliverables for this project are the source code for each of the parts of our project, the unit tests for our project, and the function-level documentation that is included in this report in C. All our deliverables will be placed into a branch in the Atlassian Stash repository for the Conduit project, which is under our liaisons' control. For redundancy, these same deliverables will be provided to our liaisons on a CD.

7.1.1 MPI Library

Code for the Conduit MPI library is found in the Conduit repository, on the `feature/hmc_conduit_mpi` branch, in the `conduit_mpi` folder.

7.1.2 I/O Library

All code for the Conduit I/O library can be found in the `hmc_2014` repository on the `mainline` branch under the `src/conduit_io` folder.

7.1.3 Timer Trees

Code for the timing tree and reduction libraries is in the `hmc_2014` repository on the `mainline` branch in the `Timing` folder.

7.1.4 Visualizer

Code for the Conduit Visualizer can be found in the `hmc_2014` repository on the `mainline` branch, in the `src/conduit_viz` folder.

7.2 Conclusion

This year, this team worked to improve the Conduit ecosystem through a number of different avenues. We interfaced Conduit with LULESH, first organizing key LULESH data in a Conduit Node and then manipulating it using Conduit MPI and Conduit I/O. This incorporation helped to ensure that our work both correct and helpful to programmers. We gathered LULESH performance data using timer trees, and tested the Visualizer on actual runtime data from LULESH. Our contributions to Conduit showed how it can be useful while also vastly enhancing its usefulness.

Future work has been addressed on a feature-specific basis. We look forward to seeing how continuing work on Conduit will expand on and complement our work, and hope that Conduit will become a widespread tool in the HPC community.

Appendix A

Glossary

LLNL Lawrence Livermore National Laboratory. See Section 1.1.

Conduit a C++ library developed by LLNL for storing and describing data. See Chapter 2.

D3.js Data Driven Documents, a JavaScript library for data visualization [4].

JSON JavaScript Object Notation, a text format for communicating structured data. JSON represents objects as nested collections of key-value pairs, which can be interpreted as appropriate in the target programming language [5].

LULESH a small physics simulation program, used within LLNL as a test bed for changes and new architectures before their use on larger programs. LULESH is an acronym for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics. See Appendix B.2.

MPI Message Passing Interface, a paradigm for parallel programming in which separate ranks of threads send and receive information in order to communicate. See Appendix B.1.

Node the primary object defined by the Conduit library. Stores data and a schema describing how that data is organized. See Section 2.1.

Schema in Conduit, a means of describing the structure and attributes of a set of data, usually in the form of a JSON string. See Chapter 2.

Silo an Input/Output library for scientific data developed by and heavily used at LLNL. See Appendix B.3.

SILEX SILO EXplorer, a GUI inspector for Silo files. Developed by and widely used within LLNL. See Appendix B.3.

VisIt an interactive visualizer for scientific data in many dimensions, capable of animating data over time. Can read in data from Silo files. See Appendix B.4.

Appendix B

Technology Background

B.1 MPI

Message Passing Interface, or MPI, is a paradigm for parallel computation [8]. A program using MPI will instantiate several processes, which may be on the same or different computers. These processes have no information about what the other processes contain and cannot directly access each others data. These nodes can only communicate information to one another through predefined MPI functions, or messages, hence the name Message Passing Interface. This allows for an increase in speed, as the number of parallel processes working on the same problem is no longer limited to the number that fit on a single computer.

MPI functions operate by sending arrays of defined data types. In most functions, the data being sent needs to be uniquely identified by the location, number, and type of data. As such, only simple, contiguous arrays of MPI types can be sent. Common functions include Send and Receive, which copy the sent array from sender to receiver, Reduce, which takes a number of arrays of the same length and type and reduces them according to some binary function (such as maximum, minimum, addition, multiplication, and so on). Scatter takes a single array and distributes pieces to a group of different processes, while Gather takes pieces from many different processes and places them in a single array. Broadcast sends an array to many different processes. There are also variants on some of the above functions that distribute compiled results, such as Allreduce, Allgather, and Alltoall, which broadcast a reduced array, broadcast a gathered array, and scatter a gathered array, respectively.

This library is heavily used in high-performance computing, as the potential parallel speedups of programs, especially physical simulations which can discretize their domain across a large number of processes.

B.2 LULESH

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics, or LULESH, is a physics simulation program written by scientists at Lawrence Livermore National Laboratory [9]. It is also used as a "mini-app", meaning a relatively small and easy to understand simulation with which features can be tested before being integrated into larger and more complex programs. LULESH operates by simulating a cube of fluid, depositing a massive amount of energy at the corner of the cube, and allowing the matter to deform according to physical principles, simulating a blast wave. It can also be run in parallel, using MPI to divide up the domain into discrete areas and simulate them in parallel.

The domain of the problem is a physics mesh, which is a series of points in a three dimensional grid pattern as well as the intervening spaces. This data is stored as a structure of one dimensional arrays, each array storing a physical value across all points, in the order the points are reduced to as a one dimensional array. LULESH has several execution options, including the number of processes to create if one is executing in parallel, whether or not to print out execution data, and whether or not to output visualization data.

Changes made to LULESH during the project were to bind the entire Domain object, which contains all execution data for the program, to a Conduit Node object, thus allowing the regular dumping of data in the form of compressed data and a schema file, so that this data could be used to resume stopped runs. Other changes include the insertion of Conduit Nodes into the communication between MPI processes, communicating through so called "ghost zones", or areas where particles in one process may affect those in another, and so their information must be transmitted to those processes. In this case, it was shown that Conduit could be used to copy the data and send it easily, without disrupting or otherwise altering the execution of the program.

B.3 Silo

Silo is an open source library used to read and write scientific data to binary files on disk. It has a C API and a Fortran API. Silo has a long history at LLNL. It was first developed in-house in the early 1990s in order to simplify data exchange between applications. It is still very tightly integrated into LLNL's workflows as a popular data format, including being used as the file format for VisIt (see B.4).

Silo stores its data in a hierarchical format that emulates a file directory. Silo supports a wide variety of specialized data structures for physics simulations. It supports a variety of integer and floating point numeric types, and vectors of said types. It also natively recognizes a number of scientific and mathematical types, including "nodes" (different from Conduit Nodes) which are vertices, "zones" which are polygons (or volumetric for higher dimensionalities) constructed of vertices, and "meshes", or multidimensional data structures, which are constructed out of zones. These meshes can have a variety of different geometries and organizations. For example, unstructured meshes do not define the shape of the polygonal faces of the mesh. Other mesh types include structured, gridless, and combinations thereof.

For more information on Silo's capabilities, please see its official documentation [11].

LLNL has a number of tools used internally to work with Silo, including the aforementioned VisIt for visualizing data stored in Silo files. Most of these other tools are built upon "browser", a command line program that can be used to navigate and inspect Silo files. These include Silex (Silo Explorer), which provides a GUI for inspecting Silo files, and `silodiff`, a command line tool for creating diffs of two silo files for comparison.

B.4 VisIt

VisIt is an open source library used to visualize scientific data. It is cross platform, interactive, and can animate data through time. VisIt can read a number of different file formats, including Silo files representing mesh data, and supports a "multiblock" format that splits parallel data across multiple files.

For more information on VisIt, please see the official website [2].

Appendix C

Documentation

C.1 Conduit MPI Library

C.1.1 Functions

Send

```
int CONDUIT_MPI_API send(Node& node, int dest, int tag,
                          MPI_Comm comm);
```

This blocking function sends a Conduit node to the specified recipient, along with a tag.

Inputs:

- `Node& node`: A reference to the Node being sent.
- `int dest`: The rank of the recipient process.
- `int tag`: The tag
- `MPI_Comm comm`: The communication channel to be used.

Recv

```
int CONDUIT_MPI_API recv(Node& node, int source, int tag,
                          MPI_Comm comm);
```

This blocking function receives a Conduit node sent by another process.

Inputs:

- `Node& node`: A reference to the Node where the user wants the receive node to be saved.
- `int source`: The rank of the source process.
- `int tag`: The tag
- `MPI_Comm comm`: The communication channel to be used.

Reduce

```
int CONDUIT_MPI_API reduce(Node& sendNode, Node& recvNode,
                            MPI_Datatype datatype, MPI_Op op,
                            unsigned int root,
                            MPI_Comm comm);
```

This function reduces a set of nodes from a number of processes according to a defined MPI operation.

Inputs:

- `Node& sendNode`: the Node that the user wishes to be reduced.
- `Node& recvNode`: the address for the resulting reduced node (only used in the root process)
- `MPI_Datatype datatype`: the datatype of every value in the node. This must be uniform across each node for the reduction to work.
- `MPI_Op op`: The binary operation that the user wants to use to reduce the nodes.
- `unsigned int root`: the process which will hold the final reduced node.
- `MPI_Comm comm`: The communication channel to be used.

Notes: Each node must have the same schema, and be of uniform type for this function to work correctly. Any structure may be used, however.

Allreduce

```
int CONDUIT_MPI_API allreduce(Node& sendNode, Node& recvNode,  
                               MPI_Datatype datatype, MPI_Op op,  
                               MPI_Comm comm);
```

This function reduces a set of nodes from a number of processes according to a defined MPI operation, then sends the resulting node to each process involved.

Inputs:

- `Node& sendNode`: the Node that the user wishes to be reduced.
- `Node& recvNode`: the address for the resulting reduced node.
- `MPI_Datatype datatype`: the datatype of every value in the node. This must be uniform across each node for the reduction to work.
- `MPI_Op op`: The binary operation that the user wants to use to reduce the nodes.
- `MPI_Comm comm`: The communication channel to be used.

Notes: Each node must have the same schema, and be of uniform type for this function to work correctly. Any structure may be used, however.

Isend

```
int CONDUIT_MPI_API Isend(Node& node, int dest, int tag,
                           MPI_Comm comm,
                           ConduitMPIRequest* request);
```

This function performs a non-blocking send.

Inputs:

- `Node& node`: A reference to the Node being sent.
- `int dest`: The rank of the recipient process.
- `int tag`: The tag
- `MPI_Comm comm`: The communication channel to be used.
- `ConduitMPIRequest* request`: A pointer to the request object for this send.

Notes: This function must be followed at a later time by `Waitsend` or `Waitallsend`, to complete the send. The recipient of the data must have a node with a matching schema, or else this function will fail.

Irecv

```
int CONDUIT_MPI_API Irecv(Node& node, int src, int tag,
                           MPI_Comm comm,
                           ConduitMPIRequest* request);
```

This function performs a non-blocking receive.

Inputs:

- `Node& node`: A reference to the Node where the user wants the receive node to be saved. Must have the same schema as the sent node.
- `int source`: The rank of the source process.
- `int tag`: The tag
- `MPI_Comm comm`: The communication channel to be used.
- `ConduitMPIRequest* request`: a pointer to the request for this receive.

Notes: This function must be followed at a later time by `Waitrecv` or `Waitallrecv`, to complete the send. The recipient of the data must have a node with a matching schema, or else this function will fail.

Waitsend

```
int CONDUIT_MPI_API Waitsend(ConduitMPIRequest* request,  
                             MPI_Status* status);
```

This function is used to wait until the send defined by a previous request completes.

Inputs:

- `ConduitMPIRequest* request`: The request object of the send the user is waiting on.
- `MPI_Status* status`: Returns the status of the send.

Waitrecv

```
int CONDUIT_MPI_API Waitrecv(ConduitMPIRequest* request,  
                             MPI_Status* status);
```

This function is used to wait until the receive defined by a previous request completes.

Inputs:

- `ConduitMPIRequest* request`: The request object of the receive the user is waiting on.
- `MPI_Status* status`: Returns the status of the send.

Waitallsend

```
int CONDUIT_MPI_API Waitallsend(int count,  
                                ConduitMPIRequest requests[],  
                                MPI_Status statuses[]);
```

This function is used to wait until all sends defined by the requests complete.

Inputs:

- `int count`: the number of requests the user is waiting on.
- `ConduitMPIRequest requests[]`: An array of previous requests from sends that the user is waiting on.
- `MPI_Status statuses[]`: An array of statuses, which will return the status of the corresponding request from `requests[]`.

Waitallrecv

```
int CONDUIT_MPI_API Waitallrecv(int count,
                                ConduitMPIRequest requests[],
                                MPI_Status statuses[]);
```

This function is used to wait until all receives defined by the requests complete.

Inputs:

- `int count`: the number of requests the user is waiting on.
- `ConduitMPIRequest requests[]`: An array of previous requests from receives that the user is waiting on.
- `MPI_Status statuses[]`: An array of statuses, which will return the status of the corresponding request from `requests[]`.

C.1.2 Structures**ConduitMPIRequest**

```
struct ConduitMPIRequest {
    MPI_Request _request;
    Node* _externalData;
    Node* _recvData;
};
```

This structure holds the request object for the underlying sends and receives in `Isend` and `Irecv`, as well as stores the data externally so that it does not leave scope and become garbage collected before the send completes.

Since `Isend` operates by creating a compacted `Node` object and sending the whole of its data, `_externalData` acts as a pointer to this heap allocated `Node` so that it can be deleted after the send completes. `_request` is the request object used by the underlying `Isend` and `Ireceive`, and `_recvData` is a pointer to the `Node` into which the user wants the new data copied, and is only used by `Irecv`.

C.2 Conduit I/O Library

The `conduit_io` library is modular, to allow for different back-end libraries to be swapped in and out. The Silo specific functionality that we added is in the header file `conduit_silo.h`. All functions are under the `conduit::io` namespace.

C.2.1 Functions

`silosave`

Definition:

```
void CONDUIT_IO_API silosave(const Node &node,
                             DBfile *dbfile,
                             const std::string &silobj_path);
```

This function stores the given Conduit node using the cold storage paradigm. It supports paths representing where in the hierarchical Silo file to store the data, allowing for a single file to hold data from multiple Nodes. The DBfile is not closed at the end of this function.

Inputs:

- `Node& node`: A reference to the Node being saved.
- `DBfile *dbfile`: A pointer to an opened DBfile.
- `const std::string &silobj_path`: The internal path inside the Silo file to use as the root for the data.

Convenience functions:

```
void CONDUIT_IO_API silosave(const Node &node,
                             const std::string
                             &file_path,
                             const std::string &silobj_path);
void CONDUIT_IO_API silosave(const Node &node,
                             const std::string &path);
```

These functions will create and open a new DBfile using the parameters to determine the file name, call the primary `silosave` function, and then close the DBfile.

Additional inputs for convenience functions:

- `const std::string &file_path`: The file path for the new Silo file.
- `const std::string &path`: The path to save the data in, consisting of the file path and the internal path concatenated. This is equivalent to `file_path + ":" + silo_obj_path`.

silo_load

Definition:

```
void CONDUIT_IO_API silo_load(DBfile *dbfile,
                              const std::string &silo_obj_path,
                              Node &node);
```

This function loads a Conduit node stored in the specified, opened DBfile using cold storage into the given node, using a Conduit generator. The DBfile is not closed at the end of this function.

Inputs:

- `DBfile *dbfile`: A pointer to a opened DBfile.
- `const std::string &silo_obj_path`: The internal path inside the Silo file at which to find the root node.
- `Node &node`: A reference to a Conduit node to populate.

Convenience functions:

```
void CONDUIT_IO_API silo_load(const std::string &file_path,
                              const std::string &silo_obj_path,
                              Node &node);
void CONDUIT_IO_API silo_load(const std::string &path,
                              Node &node);
```

These functions will open the DBfile using the parameters to determine the final name, call the primary `silo_load` function, and then close the DBfile.

Additional inputs for convenience functions:

- `const std::string &file_path`: The file path to find the Silo file.
- `const std::string &path`: The path to find the data at, consisting of the file path and the internal path concatenated. This is equivalent to `file_path + ":" + silo_obj_path`.

silo_structured_save

Definition:

```
void CONDUIT_IO_API silo_structured_save(Node &vnode,  
                                         Node &vroot,  
                                         int myRank);
```

This function will create a new object mapped Silo file. The file will be saved at the location `plot_c%d.silo` where `%d` is the current cycle of the calling program. This function will create a new Silo file and close it at the end of the operations. This function calls `silo_write_root` and `silo_write_mesh`.

Inputs:

- Node &vnode: A reference to a Conduit node structured according to the VNODE schema (see C.2.3).
- Node &vroot: A reference to a Conduit node structured according to the VROOT schema (see C.2.3).
- int myRank: The rank of the calling process for multi-core operation.

silo_write_mesh

Definition:

```
void CONDUIT_IO_API silo_write_mesh(DBfile *dbfile,  
                                     Node &node,  
                                     const std::string &subdir,  
                                     const std::string &name);
```

This function writes data to an object mapped Silo file. The file must already be opened and will not be closed at the end of the function. This function writes the numeric data and is called by `silo_structured_save`.

Inputs:

- DBfile *dbfile: A pointer to an opened DBfile.
- Node &node: A reference to a Conduit node structured according to the VNODE schema (see C.2.3).

This function is identical in use to `silo_structured_save` except that it calls the PMPIO functions to write multiblock data. This function calls `silo_write_root_pmpio` and `silo_write_mesh`.

`silo_write_root_pmpio`

Definition:

```
void CONDUIT_TO_API silo_write_root_pmpio(DBfile* dbfile,  
                                           Node &vroot,  
                                           Node &vnode,  
                                           const std::string &filename,  
                                           const std::string &meshname,  
                                           PMPIO_baton_t* bat);
```

This function is identical in use to `silo_write_root` except that it calls the PMPIO functions to write multiblock data.

C.2.3 Schemas for silo_structured_save

VNODE

```
VNODE: {
  numElem: int: number of elements;
  numNode: int: number of compute nodes;
  cycle: float: current cycle number of the computation;
  time: float: time of the computation cycle;
  Topology: {
    Unstructured: {
      hexes: int32*: connectivity for unstructured
        hex meshes;
    };
  };
  Coords: {
    x: (float32*|float64*): x coordinate data;
    y: (float32*|float64*): y coordinate data;
    z: (float32*|float64*): z coordinate data;
  };
  Fields: {
    [fieldname]: {
      data: (float32*|float64*): field data;
      centering: string: centering of the data
        must match /(zone|node)/;
    };
  };
};
```

VROOT

```
VROOT: {
  ndomains: int: number of compute nodes;
  path: string: format string representing the
               internal path for data
  Domains: {
    path: string: internal Silo path of associated data;
    type: string: the structure of the mesh data
              must match /(unstructured)/;
  };
  Expressions: {
    [exprname]: {
      definition: string: computation string;
      type: string: type of the resulting data
              must match /(scalar|vector)/;
    };
  };
};
```

The VNODE/Fields node may contain any number of children. VNODE/Expressions nodes contain special VisIt objects that can compute extra data on the fly, such as speed from positional data. The node may contain any number of children.

Future work could expand the the VNODE/Topology node to support alternatives to Unstructured data, and the VNODE/Topology/Unstructured node to support dimensionalities other than hexes. The VNODE/Coords node would need to be updated to support these additions. In addition, the VROOT/Domains node may be redundant. if so, its contents could be added to the VNODE/Topology node.

C.3 Timer Trees

C.3.1 Usage

The commands to produce timing data, run on LLNL's cab machine, were of the form

```
srunk -p pdebug -n [numberOfNodes] lulesh2.0 > [filename]
```

where [numberOfNodes] was the number of MPI processes (1, 8, or 27) and [filename] was the output file (e.g. benchmark08node_01.txt). When using MPI, the rank of the process should be set with `BlockTimer::setRank`.

The maximum depth (level of function nesting) is set by a macro in `blocktimer.h`. In the resultant timing tree, a Node's direct children are stored in a child node labelled `__children`.

To change the metrics kept track of by the timing class, one must modify `BlockTimer` (notably the static function `precheck()`), `reduce`, and the `main` function (where values are likely set manually). This is to ensure that the reduction operation exists and that any new fields are properly initialized before usage.

To add reduction functions, changes have to be made in three places:

- `blocktimer.cc`: `precheck()` needs to be updated to properly initialize any new data elements, and the destructor needs to be updated to reflect proper recording of those elements.
- `reduce.cc`: `reduce()` must be updated to do the actual reduction for any new data elements.
- `lulesh.cc`: `main()` must be updated to initialize the new data elements, so that they exist for the reduction.

C.3.2 Functions and Variables

Constructor

`BLOCK_TIMER(<name>)`

A macro that instantiates a block timer with the provided name. Starts timing when it is constructed; stops timing when it is destructed, recording and updating the time spent in the function.

setRank

```
static void BlockTimer::setRank(int rank)
```

Informs the BlockTimer class of the process rank (id). Necessary for reduceAll to function correctly.

globalNode

```
static conduit::Node globalNode
```

The timing tree. Use this for printing the timer tree and dumping it to a file.

reduce

```
void reduce(Node &a, Node &b)
```

Reduces two timing trees, a and b, into a single timing tree, and puts the result in a. If paths differ, they are merged additively (so all paths are represented). The current function keeps track of:

- maximum (___value)
- maximum process id (id)
- minimum (min)
- minimum process id (minid)
- average (avg)
- count (the number of trees that contained the node in a path) (count)

reduceAll

```
void reduce(Node &thisRanksNode)
```

Reduces the timing trees across all processes into a single timing tree and puts the result in process 0's thisRanksNode. Uses the Conduit MPI library's blocking send and receive functions.

C.4 Visualizer

C.4.1 Usage

After including `visualizer.h` in the desired program, the Visualizer can be launched by calling

```
visualize(&n);
```

where `n` is an instance of the Conduit Node class. Once this function is called in the containing program, it will print notice that the Visualizer is running to `stdout`. The user can then view the Visualizer by navigating to `localhost:8080` in a modern web browser. The Visualizer will run, blocking further execution of the calling program, until the "Quit Visualizer" button is clicked.

C.4.2 Components

Tree view

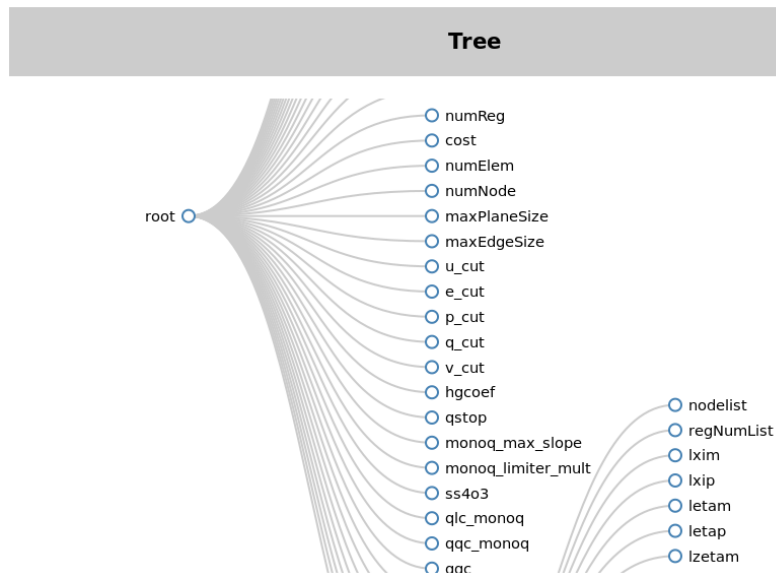


Figure C.1 The Tree view.

- The descendants of any node can be expanded/collapsed with a mouse click.

Inspector

The Inspector panel is divided into three main sections:

- Node search:** A search field containing "int", radio buttons for "name" and "type" (with "type" selected), and a "Filter" button.
- Search results:** A table with columns Path, Type, Length, and Size. It lists five results related to the search term "int".
- Value inspector:** A table with columns Index and Value, showing a list of values for the selected data member.

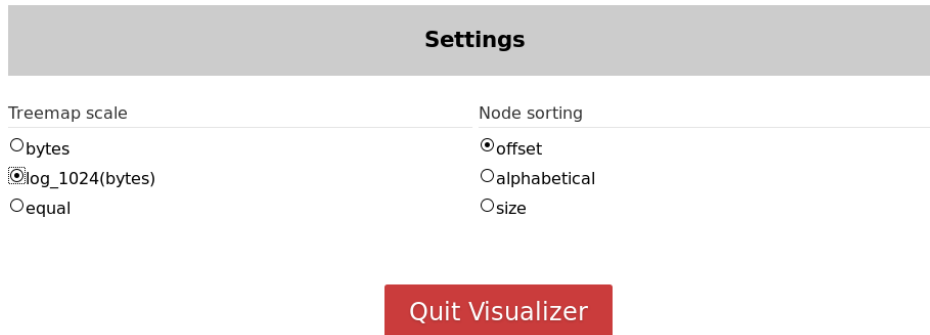
| Path | Type | Length | Size |
|-------------------|-------|--------|----------|
| /numNode | int32 | 1 | 4 B |
| /maxPlaneSize | int32 | 1 | 4 B |
| /maxEdgeSize | int32 | 1 | 4 B |
| /Elems/nodeList | int32 | 216000 | 843.8 kB |
| /Elems/regNumList | int32 | 27000 | 105.5 kB |

| Index | Value |
|-------|--------|
| 10434 | 0.675 |
| 10435 | 0.7125 |
| 10436 | 0.75 |
| 10437 | 0.7875 |
| 10438 | 0.825 |
| 10439 | 0.8625 |
| 10440 | 0.9 |

Figure C.3 The Inspector panel.

- The selected radio button determines what attributes of the data members are searched.
- The search field finds results via partial matching (searching for “foo” will match “foobar”).
- The value table shows the value of the data in the currently selected data member.
- Search results are highlighted in both the Tree and Treemap views.

Settings



| Treemap scale | Node sorting |
|--|---|
| <input type="radio"/> bytes | <input checked="" type="radio"/> offset |
| <input checked="" type="radio"/> log_1024(bytes) | <input type="radio"/> alphabetical |
| <input type="radio"/> equal | <input type="radio"/> size |

Quit Visualizer

Figure C.4 The Settings panel.

- Select a "Treemap scale" option to resize the rectangles in the Treemap according to the selected scale.
- Select a "Node sorting" option to rearrange children in the Tree, and rectangles in the Treemap, according to the selected sorting scheme.
- Click the "Quit Visualizer" button to close the Visualizer and resume execution of the calling program.

Bibliography

- [1] Google test documentation. <https://code.google.com/p/googletest/wiki/Documentation>. Accessed: 2015-04-28.
- [2] About VisIt. <https://wci.llnl.gov/simulation/computer-codes/visit>. Accessed: 2015-04-28.
- [3] European Computer Manufacturers Association et al. ECMAScript language specification, 2011.
- [4] Michael Bostock. D3.js. *Data Driven Documents*, 2012.
- [5] Tim Bray. The JavaScript object notation (JSON) data interchange format. 2014.
- [6] World Wide Web Consortium et al. HTML5 specification. *Technical Specification*, Jun, 24:2010, 2010.
- [7] Michael J Hammel. Mongoose: an embeddable web server in C. *Linux Journal*, 2010(192):2, 2010.
- [8] Rolf Hempel. The MPI standard for message passing. In *High-Performance Computing and Networking*, pages 247–252. Springer, 1994.
- [9] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 updates and changes. Technical report, Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, 2013.
- [10] Lawrence Livermore National Laboratory. SLURM at LLNL v2.3. <https://computing.llnl.gov/linux/slurm/srun.html>.
- [11] *Silo User's Guide*. Lawrence Livermore National Security, LLC, 4.10 edition, July 2014. Document Release Number LLNL-SM-654357.
- [12] Paul Masurel. fattable. <https://github.com/fulmicoton/fattable>, 2014.

- [13] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray User Group (CUG)*, 2010.
- [14] Edward M Reingold and John S. Tilford. Tidier drawings of trees. *Software Engineering, IEEE Transactions on*, (2):223–228, 1981.
- [15] Yahoo. Pure.css. <https://github.com/yahoo/pure>, 2015.

